EXHIBIT B

hClock: Hierarchical QoS for Packet Scheduling in a Hypervisor

Jean-Pascal Billaud
VMware, Inc.
billaud@vmware.com

Ajay Gulati
VMware, Inc.
agulati@vmware.com

Abstract

Higher consolidation ratios in virtualized datacenters, SSD based storage arrays and converged IO fabric are mandating the need for more flexible and powerful models for physical network adapter bandwidth allocation. Existing solutions such as fair-queuing mechanisms, traffic-shapers and hierarchical allocation techniques are insufficient in terms of dealing with various use cases for a cloud service provider.

In this paper, we present the design and implementation of a hierarchical bandwidth allocation algorithm called hClock . hClock supports three controls in a hierarchical manner: (1) minimum bandwidth guarantee, (2) rate limit and (3) shares (a.k.a weight). Our prototype implementation in VMware ESX server hypervisor, built with several optimizations to minimize CPU overhead and increase parallelism, shows that hClock is able to enforce hierarchical controls for a variety of workloads with diverse traffic patterns at scale.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Modeling techniques; D.4.8 [*Operating Systems*]: Performance—Modeling and prediction

General Terms Algorithms, Design, Management, Performance

Keywords Fair scheduling, Hypervisor, Hierarchical packet scheduling, Network QoS

1. Introduction

A typical server today consists of 16-32 cores, 128-256 GB of RAM, multiple 10GbE network interface cards (NICs) and HBAs for storage access. Larger servers bring the economies of scale to virtualized datacenters, where tens to hundreds of virtual machines (VMs) are packed on a single server, to lower cost and increase utilization.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Eurosys'13 April 15-17, 2013, Prague, Czech Republic
Copyright © 2013 ACM 978-1-4503-1994-2/13/04. . . $15.00

Hypervisors such as VMware ESX, Xen, Microsoft Hyper-V and KVM, provide good mechanisms to control resource multiplexing of CPU and memory resources between VMs. For example, CPU schedulers in most hypervisors implement some form of weight based allocation [3, 9] and handle latency sensitive [7, 30] VMs in some cases. VMware ESX server has been supporting controls like reservation, shares and limits for both CPU and memory since 2003 [29] in a hierarchical manner, which customers have found very useful. Unfortunately, these techniques are not directly applicable to network or storage IO scheduling.

mClock [12] provided similar controls for storage IO scheduling at a single host with no hierarchy. PARDA [11] and storage resource pools [13] extended these controls to a distributed environment with multiple hosts accessing the shared storage. For network bandwidth allocation at physical NICs, most hypervisors implement traffic shapers to enforce limits and/or weight-based allocation [28]. So far these controls were considered sufficient mainly because networks usually were over-provisioned and users rarely saw physical NIC saturation at servers. However, that trend is changing rapidly due to several reasons:

High consolidation ratios: Consolidation ratios per server are increasing due to higher core count and large memory sizes. In addition, two cores can easily saturate a 10G NIC using a packet size of 1 KB or higher, allowing workloads to consume a disproportionately high NIC bandwidth using small amount of CPU capacity.

Increasing storage performance due to SSDs: NFS and iSCSI based storage arrays are becoming common due to lower network costs. With adoption of SSDs in storage arrays, their IOPS are no longer limited by poor random IO performance of disks. High hit rates ($\geq 95\%$) in SSD tier are causing a significant increase in bandwidth consumption.

Increasing management traffic: Management traffic is consuming significant bandwidth in virtual environments due to primitives like vMotion, storage vMotion, which allow live migration of virtual machines and virtual disks respectively [21].

Converged architectures: Converged storage and compute architectures are becoming common, where local storage across servers is stitched together to provide a shared storage device. Several companies such as Nutanix [15], Sim-

plivity [20], HP [14] are offering products that obviate the need for a separate storage area network and use local NIC bandwidth for replication and IO access from remote nodes.

Increased network consumption makes it critical to provide better scheduling at physical NICs on hosts. The QoS model supported by the packet scheduler needs to be rich, flexible, simple to use and efficient to enforce. Based on various customer use cases in datacenters and cloud hosting environments, we have realized that the model should support hierarchical allocation, support minimum guarantees, support rate limit along with prioritized allocation of spare bandwidth based on weights (Section 2).

In this paper, we present the design and implementation of a network IO scheduler, called hClock . hClock can support all three controls in terms of minimum reservation (R), maximum limit (L) and shares (S) for proportional allocation, in a hierarchical manner.

We compare this resource allocation model to several existing models such as weight-based, rate-limiting, service curves [5, 18] and hierarchical service curves [25] in Section 2. The details of hClock algorithm are presented in Section 3 followed by a walk through of hClock scheduler using an example in Section 4.

We also implemented several optimizations to increase the parallelism and reduce CPU overhead, to make our solution practical in a highly performant hypervisor (Section 5). Our prototype implementation inside VMware ESX hypervisor [27] shows that hClock is able to provide these controls in an efficient and work-conserving manner (Section 6). Finally, we conclude with the hope that hClock can be used as a key building block in providing end-to-end network resource management [24] (Section 7).

2. Background and Motivation

In this section, we provide an overview of various resource allocation models that have been proposed in the literature and make a case for why we chose to implement our specific model instead of using one of the existing ones.

(1) Proportional share allocation: Shares or weight based proportional allocation is one of the oldest and most widely used model for resource allocation. This model is simple to implement and various approaches have been proposed in the literature, such as WFQ [6], WF^2Q [1], DRR [19], SFQ [10] and several others [4, 8, 22, 23, 26]. These provide similar high level controls although with different worst case delay bounds, worst case fairness guarantees and implementation complexities such as $O(\log\ N)$ for SFQ vs. $O(1)$ for DRR.

The main concern with only using shares is that the allocation of one VM is affected by other VMs and can get reduced as more VMs are started on the host. It is also hard to provide a lower bound (*i.e.* reservation) using shares, due to available bandwidth fluctuation at the NIC and the need to adjust shares every time a new VM is powered-on or migrated. Furthermore, typically reservations don't always add up to the NIC bandwidth and one may want to allocate spare capacity differently. But using shares to provide reservations couples these two controls together. The complexity of changing shares every time new VMs are added or removed is also cumbersome in highly dynamic virtual environments.

(2) Shares with reservations and limits (no-hierarchy): Adding reservation and limit as additional controls provides flexibility in terms of handling various use cases. Reservations help provide peace of mind and strict performance guarantees in a cloud environment and alleviate the need of perpetual QoS controls adjustment as VMs come and go. Limits are very useful for cloud service providers to bound the usage of per-tenant network bandwidth in a multi-tenant cloud. This allows for proper charge-back based on allocation and strict performance isolation. Limits can also avoid potential malicious behavior from VMs that can try to dominate a shared resource. Typically, the well known models of leaky bucket and token bucket are used for for rate limiting.

There are two key limitations of this model. First, the resource settings need to be supplied for each VM. This presents manageability issues where a server may have different traffic classes and VM traffic is just one of them. Given few hundred VMs on the server, it is undesirable to specify controls for each VM. Second, setting individual controls does not allow desired statistical multiplexing. For instance, one may want to limit a class of traffic instead of setting hard limits on individual components.

(3) Service curves: Service curves is a powerful resource allocation model initially proposed by Parekh and Gallaghar [16, 17], and later extended by Cruz et. al. [5, 18] to decouple latency and throughput requirements. Different service curves per flow can provide better latency properties while doing proportional share allocation. Figure 1 shows a traffic flow with bursty arrivals and how concave and convex service curves with two slopes can be used elegantly to trade-off latency between flows. Share-based allocation is similar to a single straight line service curve where the slope of line is equal to the share-based allocation.

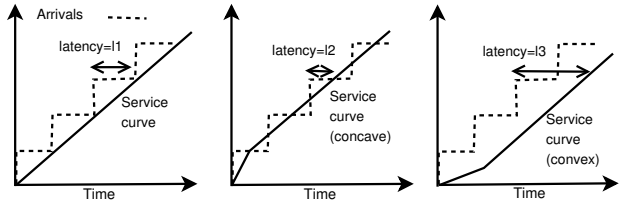

Figure 1. Service curve based allocations with equal rate = slope of second line. A concave service curves reduces latency and a convex curve increases latency: $l2 < l1 < l3$

(4) Hierarchical service curves: Stoica et al. [25] extended the notion of service curves to do hierarchical allocation. The authors showed a valuable result that it is not

Algorithm class	Weighted allocation	Reservation support	Limit support	Hierarchical allocation	Admission control
Proportional Sharing (PS) Algorithms	Yes	No	No	No	No
PS + Reservations,Limit	Yes	Yes	Yes	No	Yes
Hierarchical Service Curves	Yes	No	Yes	Yes	No
mClock	Yes	Yes	Yes	No	Yes
hClock	Yes	Yes	Yes	Yes	Yes

Table 1. Comparison of hClock with existing scheduling techniques

possible to support non-linear service curves at various levels of hierarchy and presented a solution that supported such curves only at leaf nodes. The internal nodes simply get shares based allocation.

We did not use the hierarchical service curves due to a couple of reasons. First, they don't support controls like reservation and limits explicitly. It is not obvious how one can add these controls. Second, we need to be able to do admission control when VMs are added to the system and it is not clear how to do that while using hierarchical service curves. Finally, the overall behavior is unclear when different service curves are introduced in a hierarchical manner. Figure 2(a) shows a case where service curves supporting latency sensitive applications are put as a child to a non-latency sensitive curve and vice versa. In such cases, the actual use case and behavior of the system is not clear.

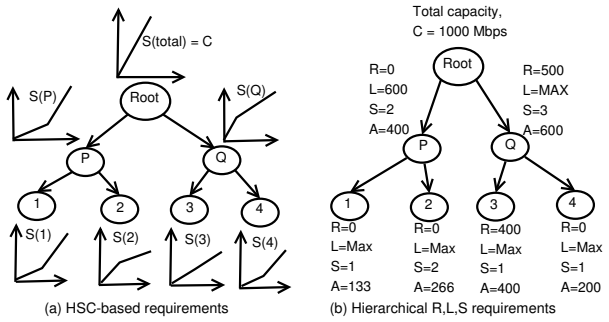

Figure 2. (a) Hierarchical service curves vs. (b) hierarchical tree with R,L,S settings. The actual allocated capacity is denoted by A, using MAX(R,S) semantics.

(5) Hierarchical shares, reservations and limits (Our approach): This model supports all three controls in a hierarchical manner as shown in Figure 2(b). A valid setting of these controls must satisfy the following two simple constraints: (1) parent reservation has to be greater than or equal to the sum of its children reservation and (2) a node's limit cannot be less than its reservation. Reservation values are also used to do admission control. This is the model implemented by hClock . The allocation needs to satisfy the following properties at every level: (1) Every node gets at least its reservation, (2) Every node gets at most its limit. The allocation of spare bandwidth left after satisfying reservations, can be done using shares with two different semantics, which we call as **MAX(R,S)** and **SUM(R,S)**.

In MAX(R,S) case, the goal is to do *overall allocation in proportion to shares* while meeting the reservation and limit constraints. For example, the root capacity of 1000 Mbps in Figure 2(b) is divided in proportion to shares $2:3$ among the nodes P and Q, while also satisfying the R and L constraints. Similarly 400 Mbps at P is further divided among its children in ratio of their shares $1:2$. However, 600 Mbps at Q is divided among the children as 400 and 200, because even though they should get 300 Mbps each based on the equal shares, the node 3 has a reservation of 400 Mbps that needs to be met. Node allocations based on MAX(R,S) are shown in Figure 2(b).

In SUM(R,S) semantics, the capacity at a parent is first allocated based on children reservations and *only the remaining capacity* is further allocated *based on the share values* while respecting limit constraint. So the allocation to node P will be $0 + (1000 - 500) * \frac{2}{5} = 200$ and the allocation to node Q will be $500 + (1000 - 500) * \frac{3}{5} = 800$. Thus a node gets its reservation plus some of the remaining capacity based on its shares. Both of these semantics can be configured using different shares settings to provide similar allocations. It is hard to argue for one semantics to be strictly better than the other. So we decided to implement both in our design and make it configurable. Table 1 shows a summary of comparison between various techniques.

3. Hierarchical Packet Scheduler

In this section, we present the design of hClock algorithm. Figure 3 presents a system overview, with 6 VMs connected to a virtual switch and the location of hClock in the networking stack. It also shows an example of resource hierarchy with two levels. Each VM is denoted by a queue q, with three resource settings: bandwidth reservation r_q, upper limit l_q and shares s_q for spare capacity allocation. Bennett and Zhang [2] presented an elegant solution providing hierarchical shares-based allocation by using a one level PFQ (packet fair queuing) algorithm and extending that to build H-GPS. In a similar manner, we start with mClock [12], that provides all of these three controls at a single level and extend that to provide a hierarchical solution.

We first present a brief outline of mClock algorithm that supports the same three controls per queue in a non-hierarchical manner. mClock uses three separate tags per queue q. These tags namely, reservation tag (R_q), limit tag

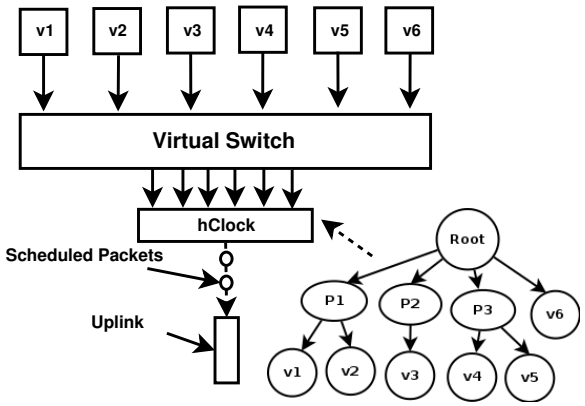

Figure 3. hClock as part of hypervisor networking stack

Symbol	Meaning
T	Current real time
q	A queue regardless of its type
s_q	Shares of queue q
r_q	Reservation setting of queue q
l_q	Limit setting of queue q
S_q	Proportional shares tag of queue q
R_q	Reservation tag of queue q
L_q	Limit tag of queue q
K, M, G	10^3, 10^6 and 10^9

Table 2. Symbols used and their descriptions

(L_q) and shares tag (S_q) are used to track the allocation based on reservation value (r_q), limit value (l_q) and shares value (s_q) respectively. These tags are incremented based on corresponding r, l and s values whenever each IO is scheduled. All of these tags are in real time domain to track the fulfillment of reservations and limits by comparing the corresponding tags with the real time.

At the time of scheduling an IO, the scheduler checks to see if the smallest reservation tag among all queues is not higher than the current time. If that is true, the scheduling is done based on reservation tags and an IO from the queue with smallest reservation tag is selected. Otherwise, the scheduling is done based on share tags and a request from a queue with the smallest share tag is chosen. For either case, a check is made to make sure that limit tag is smaller than current time (*i.e.* $L_q \leq T$), as a filter.

Even though hClock uses mClock as a building block, simply extending the concepts to be hierarchical where similar decisions are made at every level in the hierarchy didn't quite work. In the following, we present various components of hClock and the reasons we moved away from a naive hierarchical version of mClock. Table 2 presents the notation used in this paper for quick reference.

3.1 hClock Algorithm

There are three main components of the hClock algorithm that we explain next: (1) Queuing of incoming requests, (2) Request scheduling, and (3) Request billing.

Incoming requests: A request can only arrive at a leaf queue in the hierarchy. The internal nodes merely represent a grouping construct to specify controls but do not have a physical queue. We maintain three tags and activity status for the internal nodes as well as leaves. The three tags keep track of reservation, limit and shares controls for every node and the activity status represents whether a node has any pending requests.

When a new request arrives in a queue q, we first check the activity status. If the queue is already active, the request is simply added to the end of the queue. The more interesting case is when q is idle. If so, we need to activate the queue which allows it to be considered during next scheduling instance.

In a non-hierarchical setting it simply means adding it to the set of currently active queues. In a hierarchical setting, it is more complicated since a leaf queue activation might mean the activation of an entire branch all the way up to the root node. hClock algorithm activates all the queues starting from the original leaf queue q to the root until a non-idle node is encountered. Note that an internal node is considered idle if none of its children is active. Furthermore, during this process the scheduler needs to initialize the tags on every node along the activated path based on the QoS settings. Regardless of the nature of node the tags are initialized the same way.

The reservation tag (R_q) and limit tag (L_q) are maintained on the real time scale, denoted as T and are initialized using the maximum of the previous value and current time. This is done to synchronize a newly active queue with the currently active ones, so that it cannot starve the currently active queues due to an old reservation tag. The following equations are used to update these tags at every level of the hierarchy:

$$R_q \leftarrow \max\{R_q, T\} \quad (1)$$

$$L_q \leftarrow \max\{L_q, T\} \quad (2)$$

Unlike mClock, we use virtual time for share tags (S_q) instead of real time. This improves the implementation of shares tag synchronization. In mClock, the newly active queue is given the share tag equal to the current real time and all other queues are adjusted to bring the minimum share tag to the current time. In hClock , we simply initialize the shares tag of the newly active queue to be the minimum shares tag among its active siblings *e.g.* active nodes that share the same parent in the hierarchy. This provides significant performance boost by converting an $O(n)$ operation to an $O(1)$ operation. The following equation is used at each level in the hierarchy to update share tags:

$$S_q \leftarrow \max\{S_q, \min(\forall_{(r \in sibling\ active\ queues)} S_r)\} \quad (3)$$

Every internal node in the tree maintains minHeap structures per tag to efficiently select a queue for scheduling. As

queues get activated, the scheduler places them into the relevant minHeap so that the next scheduling cycle can consider them. While hClock keeps a minHeap per internal node to track limit and shares tags of the children, it maintains a **single global minHeap for reservation tags** for all nodes in the tree. This significantly improves the handling of reservations and helps in meeting reservations at much finer granularity. We show the problem with keeping a separate minHeap for reservation at every level, using a detailed example in section 3.2.

Hence upon activation, the scheduler inserts a node's S_q and L_q into its parent's shares and limit minHeap and may also insert R_q in the global reservation minHeap if a minimum bandwidth is configured. Algorithms 1 and 2 show the pseudo code of the enqueue function of hClock . In that, adding a queue to a minHeap is equivalent to marking it active for next scheduling instance.

Algorithm 1: Enqueue packet

```
1  EnQueuePacket (queue q, packet p)
2      append packet to the q
3      if q was idle then
4          ActivateQueue(q)
5      ScheduleRequest()
```

Algorithm 2: Activate Queue

```
1  ActivateQueue (queue q)
2      let T be the current real time
3      let p_q be the parent of q
4
5      R_q = max{R_q, T} /* reservation tag */
6      L_q = max{L_q, T} /* limit tag */
7      S_q = max{S_q, min(∀_{r∈siblings} S_r)} /* shares tag */
8      add q to global reservation minHeap
9      add q to p_q's shares and limit minHeaps
10     if p_q was idle then
11         ActivateQueue(p_q)
```

Request scheduling: hClock maintains reservation and limit tags in real time domain and shares tag at each node in a virtual time domain. The scheduling starts at the root node and the scheduler first checks if there are any nodes eligible based on reservation, whose R_q and L_q are smaller or equal to the current time T. If the set of such nodes is not empty then the one with smallest R_q is selected and all of its ancestors are marked as scheduled based on reservation. Otherwise the root node is chosen.

Starting at the node chosen in the previous step, hClock descends the tree to find nodes based on shares. For each internal node encountered on the way down, the child node with smallest S_q and that also has $L_q \leq T$ is selected for further investigation. Note that shares tags are only compared at a single level in the hierarchy among the sibling nodes. Eventually, this process stops once a leaf node is reached.

Finally, the number of outstanding bytes from scheduler is bounded by a $max\_inflight$ parameter, to control the NIC utilization. This is a function of the link capacity and the interrupt coalescing time of the device. For instance, a 10 $Gbps$ link sends 1250 bytes in $1us$. If the interrupt coalescing time is set to every $20us$ then the $max\_inflight$ should be set to 25 KB, which is the amount of data sent within one interrupt cycle. Algorithm 3, 4 and 5 present the pseudo code for request scheduling.

Algorithm 3: Scheduling Process

```
1  let max_inflight = maximum allowed outstanding bytes
2  let inflight be the current outstanding bytes
3
4  ScheduleRequest ()
5      while inflight ≤ max_inflight do
6          q = FindEligibleQueue()
7          p = DeQueuePacket(q)
8          inflight += packetLength of p
9          send p to the device for transmit
```

Algorithm 4: Queue Eligibility

```
1  FindEligibleQueue ()
2      let E be the set of queues with R_q and L_q ≤ T
3      if E is not empty then
4          let q be the queue with the minimum R_q
5          flag q and ancestors as eligible for reservation
6      else
7          let q be the root of the hierarchy
8      return FindEligibleQueueBasedOnShares(q)
9
10 FindEligibleQueueBasedOnShares (queue q)
11     if q is a leaf then
12         return q
13     let E_c be the set of q's children, whose limit tag ≤ T
14     if E_c is empty then
15         return nil
16     let c be the child queue of q with minimum S_c
17     return FindEligibleQueueBasedOnShares(c)
```

Algorithm 5: Dequeue Packet

```
1  DeQueuePacket (queue q)
2      pop first packet p from the queue q
3      let packetLength be the length of the packet
4      BillHierarchy(q, packetLength, q is idle?)
5      return p
```

Request billing: Once a packet is selected for transmission a billing phase is started to ensure that everybody in the hierarchy, from root to the selected queue pays for the packet. The billing process is done in a bottom-up manner. At each node, the scheduler decides whether this particular node should be billed based on reservation or proportional shares. If the scheduling is done based on shares, the shares tag is moved forward based on the packet length and the shares of the node (denoted as s_q), using the following equation:

$$S_q \leftarrow S_q + packetLength/s_q \tag{4}$$

The above operation is repeated until the root or a queue picked based on reservation is met. If a queue elected based on reservation is encountered then for that queue and its ancestors only the reservation tag R_q is moved ahead, using the following equation:

$$R_q \leftarrow R_q + packetLength/r_q \tag{5}$$

The reason the ancestors are also billed for reservation is to enforce the hierarchical semantic of reservation. This gives us SUM(R,S) semantics. For MAX(R,S), we update the share tags in addition to the reservation tags at each level during reservation billing. Accounting for limit is simpler since regardless of the tag used for scheduling, the limit tag L_q is always moved forward using the following equation:

$$L_q \leftarrow L_q + packetLength/l_q \tag{6}$$

Intuitively the real-time based tags, R_q and L_q, are advanced based on the time span required to send the payload given the minimum guarantee r_q and maximum constraint l_q. For instance if a VM v_i requires a minimum guarantee of 100 Mbps then it essentially needs to send 1250 bytes every 100 us. Therefore, if v_i constantly sends 8 KB packets then its reservation tag R_q keeps jumping $640us$ forward and its next packet will only be picked up once the real-time clock is advanced by that value. Algorithm 6 presents the pseudo code for the billing mechanism.

Algorithm 6: Request billing process

```
1  BillHierarchy (queue q, int packetLen, bool isPending)
2      let p_q be the parent node of q
3      if q was eligible based on reservation then
4          R_q+ = packetLength/r_q /* reservation tag */
5          if use MAX(R,S) semantics then
6              S_q+ = packetLen/s_q /* shares tag */
7      else
8          S_q+ = packetLength/s_q /* shares tag */
9      L_q+ = L_q + packetLen/l_q
10     if isPending = false then
11         remove q from all minHeaps
12     BillHierarchy(p_q, packetLen, p_q is idle?)
```

3.2 Why Single Global minHeap for Reservation

In theory, one can keep one minHeap for reservation at each node in the hierarchy, similar to shares and limit. However, we found that it doesn't quite work in practice due to variable packet sizes. Consider a scenario with an internal node n with reservation of 100 Mbps and two children v_1 and v_2 with reservations 50 Mbps and 0 respectively. The internal node n also has some siblings without reservation.

At $T = 0us$, the scheduler starts at the root and picks n with reservation tag $R_n = 0$. Starting at n, the scheduler applies the same logic and selects v_1 with reservation tag $R_{v1} = 0$. At that point a packet of 8 KB is dequeued from v_1's queue. At 50 Mbps it would take $1280us$ to send such a packet size therefore v_1's reservation tag R_{v1} is advanced to $1280us$ and reservation tag of parent (node n), R_n is advanced to $640us$.

After sending a couple of packets based on shares, at $T = 640us$, the scheduler needs to satisfy the reservation constraint for n again. Looking at n's children, v_1 is not eligible for reservation-based scheduling because its reservation tag R_{v1} is still greater than T. Therefore, the scheduler falls back to proportional shares scheduling and picks v_2. It turns out that v_2 is throughput heavy and sends 64 KB packets. After billing v_2 for shares-based scheduling and incrementing its shares tag, the billing process now will increase the reservation tag of the parent n for its reservation usage. At 100 Mbps it would take $5120us$ to send a 64 KB packet and therefore the reservation tag R_n is advanced to $5760us$. Due to this, v_1 is finally serviced as n's reservation becomes eligible again after real time crosses $5760us$. Unfortunately v_1's minimum guarantee constraint is way past due by then.

The main problem here is that the scheduler did not have an exhaustive view of the reservations at any time T. One can try to fix this by maintaining a separate reservation tag at each parent to track the minimum reservation tag among its active children. But at that point it is simpler to have a global minHeap to keep track of minimum reservation tag where each node with a minimum guarantee is represented in a flat way. After all, the real time T is common to all nodes in the hierarchy. Now it is impossible for v_1 to be serviced late just because its sibling tends to send large packets.

4. hClock Example

This section shows the working of hClock using the example hierarchy shown in Figure 4 using the SUM(R,S) semantics. For the sake of simplicity each node in the hierarchy is constantly backlogged with a steady stream of 1 KB packets. A link capacity of 250 Mbps is assumed and scheduling cycles, denoted as C_i, are spaced out by $32us$ based on the theoretical time span needed by the link to transmit a 1 KB packet. To ease the understanding of this example, Figures 5, 6 and 7 show the progression over time of the reservation, limit and shares tags respectively. The progression of each tag is represented by a curved arrow annotated with the cycle

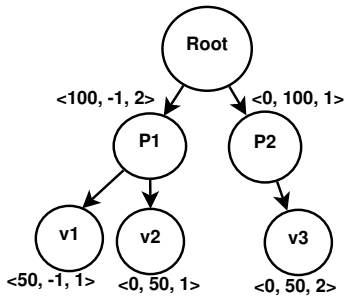

Figure 4. An example hierarchy where QoS controls are defined using the $< R_{Mbps}, L_{Mbps}, S >$ format. No limit setting is denoted as -1

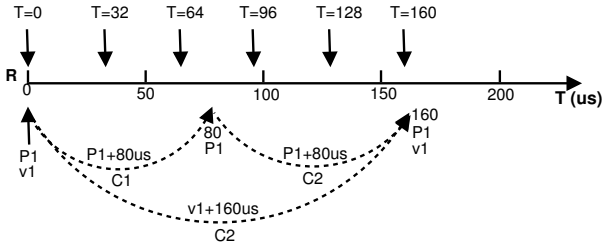

Figure 5. Real time line tracking reservation tags.

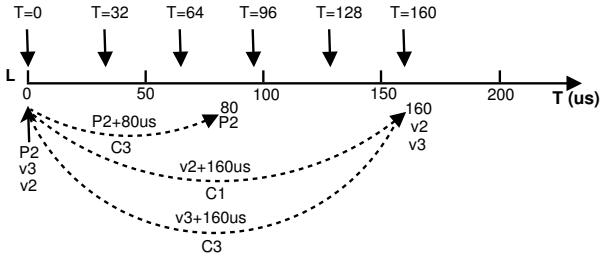

Figure 6. Real time line tracking limit tags. Although hClock maintains a minHeap for limit on each node, we are showing a single timeline for the ease of exposition

C_i responsible for the scheduling decision and the amount of real or virtual time billed for the relevant elected node.

At $T = 0us$, all tags are at 0 and the first scheduling cycle C_1 starts. Based on Algorithm 4, hClock looks first at the global reservation timeline to get a set of nodes whose R_q and L_q are $\leq T$. With p_1 and v_1 both eligible, the scheduler picks p_1 after a random tie break and switch to its shares based scheme, since p_1 is an internal node, to look for the children of p_1 whose $L_q \leq T$. In this case, both v_2 and v_1 are eligible and the scheduler chooses v_2 after a random tie break as $S_{v1} = S_{v2}$. With v_2 being a leaf, the election is done and the scheduler can start the billing process. Per Algorithm 6, v_2 is billed based on shares and S_{v2} jumps by 1000 units. The limit tag L_{v2} is moved $160us$ ahead as well based on its 50 Mbps limit. Moving up to p_1, which was elected based on reservation, R_{p1} is moved $80us$ ahead

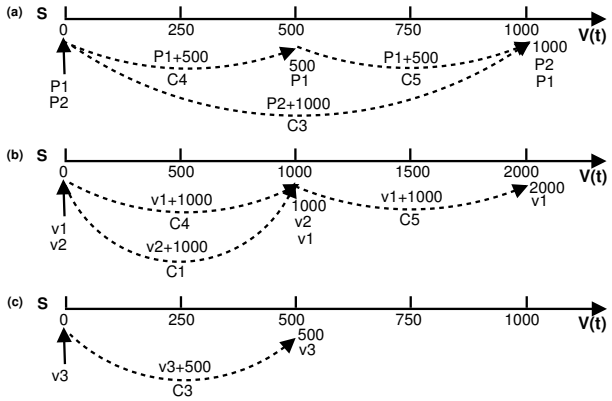

Figure 7. Virtual time line tracking shares tags at (a) root (b) p_1 and (c) p_2.

based on its 100 Mbps reservation.

At $T = 32us$, next scheduling event C_2 happens and the scheduler picks v_1 due to its reservation tag of 0. Per Algorithm 4, v_1 being a leaf there is no point going any further. From there, the scheduler charges R_{v1} of $160us$ based on its 50 Mbps minimum bandwidth requirement and leaves L_{v1} unchanged since there is no limit configured. Per Algorithm 6, once the algorithm finds a node elected based on reservation every single of its ancestors are to be billed based on reservation as well. Therefore R_{p1} is moved $80us$ ahead.

At $T = 64us$, C_3 starts and none of the nodes is eligible for reservation because all reservation tags are greater than $64us$. Hence, per Algorithm 4, the scheduler looks for a leaf eligible based on shares starting from the root and moving down one level at a time using share tags. In this case, both p_1 and p_2 are eligible based on shares and $L_{p2} \leq T$. hClock picks p_2 after a random tie break. From there, v_3 with $L_{v3} \leq T$ is the only remaining candidate which ends the election process. With v_3 being elected based on shares, S_{v3} is moved 500 units ahead based on a share value of 2 and L_{v3} jumps by $160us$ based on the 50 Mbps limit set. Since p_2 was also elected based on shares, S_{p2} is charged 1000 units while L_{p2} is billed $80us$ due to its 100 Mbps limit.

At $T = 96us$ and $T = 128us$, once again none of the nodes is eligible for reservation. Starting from the root, the scheduler picks p_1 based on shares since $S_{p1} < S_{p2}$. At last, the scheduler figures that v_1 must be the elected leaf node for C_4 and C_5 since v_2 is not eligible with $L_{v2} > T$. Finally, based on Algorithm 6, during C_4 and C_5, v_1 and p_1 are billed based on shares and S_{v1} and S_{p1} jump respectively 1000 and 500 units ahead.

At $T = 160us$, the scheduler state is back to that of C_1 and therefore the bandwidth distribution pattern observed from C_1 to C_5 is expected to repeat itself endlessly given that the tie breaks yield the same outcome. If the tie breaks yield different decisions, the bandwidth allocation will still be similar, although the order in which leaf queues are elected

may be different at small scale.

Based on 250 Mbps bandwidth, in $160us$, the scheduler should have sent 5 KB which turns out to be accurate since we had 5 scheduling cycles each sending a 1 KB packet. Next we compare the bytes sent by each node during this timeframe to the expected bytes per node.

p_1: p_1 should get $100 + (250 - 100) * \frac{2}{3} = 200$ Mbps, which is equivalent to 4 KB in the first $160us$. This is indeed the case since packets were sent on behalf of p_1 during C_1, C_2, C_4 and C_5.

p_2: Based on shares, p_2 should get $(250 - 100) * \frac{1}{3} = 50$ Mbps, which translates to 1 KB in $160us$. This does happen since a single packet was sent on behalf of p_2 during C_3.

v_1, v_2: with a 50Mbps reservation v_1 is guaranteed to send 1 KB. Based on equal shares, v_1 and v_2 should get 75 Mbps from the remaining 150 Mbps at p_1. But v_2 is capped at 50 Mbps, so 100 Mbps are allocated to v_1. Therefore, v_1 should get a total of 150 Mbps or 3 KB and v_2 should only get 50 Mbps or 1 KB. Looking at the previous scheduling scenario, v_1 sent a total of 3 KB indeed during C_2, C_4 and C_5 while v_2 only sent 1 KB during C_1.

v_3: given that p_2 bandwidth allocation and v_3 limit both equal to 50 Mbps that is as much as what v_3 can possibly get. Hence v_3 should have sent a single 1 KB packet from C_1 to C_5. Indeed, C_3 was the only cycle in which v_3 sent a packet.

Given that the allocation from C_1 to C_5 demonstrate a proper enforcement of resource controls and this pattern will repeat over time, we can see that the algorithm provides the expected allocation to all nodes. Although we don't have a formal proof, we hope that this example provides an intuitive understanding of how hClock handles all these controls by switching between reservation and shares based scheduling, while enforcing limit constraint.

5. Efficiency Optimizations

We implemented several optimizations to minimize the runtime overhead of hierarchical scheduling. We discuss some of them in detail next.

5.1 Handling Ineligible Children Due to Limits

With hierarchical limits we found an efficiency issue that impacted performance but not correctness. Consider an example with a parent node n with a limit set to 100 Mbps and a single child v_1 with a limit of 50 Mbps.

At time $T = 0us$, the limit tags of parent n and v_1 are zero, *i.e.* L_n=L_{v1}=0. The scheduler goes down to v_1 and dequeue a 8 KB packet. On the billing bottom-up process, L_{v1} is moved ahead to $1280us$ and L_n is moved ahead to $640us$. At $T = 640us$, the parent n is eligible for proportional shares scheduling. Unfortunately the scheduler won't find any queue eligible underneath since L_{v1} has not become eligible yet. Getting such empty queues is fairly costly since it can make the scheduling algorithm run linearly with respect to the number of nodes in the hierarchy as opposed to logarithmically. To handle this, we make the parent limit tag equal to the maximum of its own limit tag and the lowest limit tag among its active children. If a child does not have any maximum bandwidth constraint then its limit tag is considered 0.

Going back to the previous example, the scheduler would not pick n in the first place since its L_n would be set to L_{v1}. Thus our limit tag setting formula for an internal node when it becomes active is:

$$L_n \leftarrow \max\{L_n, L_c, T\} \tag{7}$$

Here L_c is the minimum L_q among all the active children of node n. Similarly, the billing of limit tag at an internal node is done using:

$$L_n \leftarrow \max\{L_n + packetLength/l_n, L_c\} \tag{8}$$

Conveniently this also takes care of the case where the scheduler elects an internal node based on reservation while none of its children is eligible due to limit constraint since Algorithm 4 looks at L_q before making any decision.

5.2 Fine-grained Locking

Pumping networking traffic in a virtualized server requires two parties to work side by side without invading each others space: VMs that push packets in a queue and hypervisor that drains them. Minimizing the dependence between these two parties is critical. Achieving this meant optimizing our locking model.

A naive locking model is to have a global lock around the hierarchy so that everybody waits for their turn to queue their packets. It also means that the scheduler needs to wait for its turn to actually drain packets. We found the performance penalty of this model to be too high.

In our approach, we implemented a dedicated software queue for each VM virtual NIC in the system so that a VM can get in and get out as fast as possible. More specifically, we use a locking model where every node in the hierarchy has its own lock. The only point of contention occurs once the scheduler dequeues and a VM enqueues packets on the same queue concurrently in which case the queue lock needs to be held exclusively. Based on our testing this contention is quite minimal though.

In a flat system, this model is not too hard to implement. However, it proved challenging to achieve in a hierarchical system with multiple levels. This is because when a VM or the scheduler added or removed a packet from a queue, we had to make a decision about activation or deactivation of the corresponding queue and its parent nodes all the way up to the root in a non-atomic manner. It basically means that a node in the hierarchy can be in the process of being deactivated as well as activated in parallel by two different threads. This can cause inconsistency in the system.

Our implementation addresses this issue by adding a two steps activation mechanism. The idea is that each internal

node comes with a dedicated list for pending activations. When a VM is being activated, after queuing its first packet, it simply pushes itself in its parent activation pending list. It is the scheduler's responsibility to go over this activation pending list and integrate the queues in the relevant min-Heap. The scheduler does this integration in a lazy manner which means that it will integrate children node in pending activation state only on the branch chosen for scheduling as opposed to integrating everybody upfront. This pending activation list also led to an optimization related to limit handling since the scheduler simply doesn't integrate the children from an activation list whose limit tag is greater than the current time.

5.3 Batching Small Messages

Scheduling becomes a challenge at very high packet rates. For small message size such as 256 bytes, it is too costly to go through the hierarchy and update all the tags involved on a per packet basis. We implemented batching on transmission, which means that when the scheduler found an eligible leaf queue it will drain packets from the same queue until it reaches 10 KB size worth of payload. Such degree of batching doesn't really impact fairness at the macro scale but it boosted performance quite significantly.

More importantly, our experiments revealed that small packets and very high reservation can disturb the fairness of the scheduler due to integer-based accounting in the kernel. Consider a leaf queue v_1 that sends a constant stream of 1 KB packets with a 6 Gbps reservation set. Without a batching mechanism v_1 might actually get a higher reservation depending on the time granularity of the tags. On VMware ESX server hypervisor [27] the time is defined at the microsecond scale.

At 6 Gbps it would take $1.33us$ to send a 1 KB packet but due to the integer round-up for each 1 KB, R_q only moves $1us$ forward. Hence v_1 becomes eligible every $1us$ which results in a bandwidth allocation of 8 Gbps. Batching 10 KB at a time mitigates the impact of this timing approximation. Indeed in this case it would take $13.3us$ to send a 10 KB packet and R_q moves $13us$ ahead instead. Effectively v_1 ends up sending 10 KB every $13us$ resulting in an actual bandwidth of 6.15 Gbps or 2% of error which is much more acceptable. Note that this error reduces significantly as the requested reservation gets smaller.

6. Experimental Evaluation

In this section, we present a detailed evaluation of our hClock prototype in the VMware ESX server hypervisor [27]. We examined the following key questions in our experiments: (1) Why do we need hierarchical QoS? (2) How well does hClock enforce hierarchical reservation, limit and shares for diverse traffic types? (3) What is the overall CPU overhead of hClock scheduler as compared to a simple FIFO scheduler and how does it scale with the number of VMs and

	workload	message size	socket size
v_1	tcp	64KB	256KB
v_2	tcp	256B	8KB
v_3	tcp	8KB	32KB
v_4	udp	1KB	256KB
v_5	tcp	32KB	128KB
v_6	tcp	16KB	64KB

Table 3. VM workload characteristics and parameters

(4) How does hClock handle over-committed reservations?

6.1 Experimental Setup

We implemented hClock as a kernel module that hooks itself in an uplink data path between a Virtual Switch and a Physical adapter. For experiments, we used two HP ProLiant ML350 G6 servers, each with two quad-core Intel Xeon 2.40GHz processors, 12GB of RAM and an Intel 82598EB 10G NIC card with a interrupt coalescing time of $33us$. One host was dedicated for networking traffic transmission and it ran VMware ESX server hypervisor with our prototype and six one virtual CPU VMs. Each VM ran Ubuntu Maverick (10.10) with a vmxnet3 virtual network adapter. The second host ran a native Ubuntu Precise (12.04) and was dedicated for networking traffic reception.

6.2 Need For Hierarchy

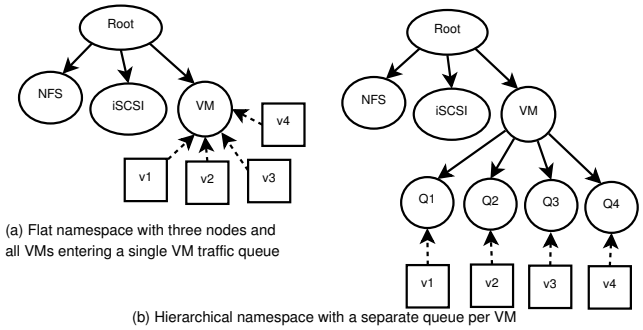

Figure 8. (a) A flat namespace with three traffic classes and a single queue shared by all the VMs v_i (b) Hierarchical namespace with dedicated queue per VM v_i.

Network administrators typically manage well-known aggregated traffic classes such as NFS, iSCSI, VoIP, VM traffic etc. without worrying too much about the individual consumers, *e.g.*individual VMs or virtual NICs. This results into dedicating a single class of service for an entire group of VMs or traffic flows.

To showcase the need for hierarchy we start with a simple set up with three different traffic types: NFS, iSCSI and overall VM traffic using the native SFQ [10] implementation built in the VMware ESX server hypervisor. We partition the overall bandwidth among these three in equal ratio. Each of the NFS and iSCSI traffic is emulated with a single VM denoted by v_{nfs} and v_{iscsi} respectively. We then used

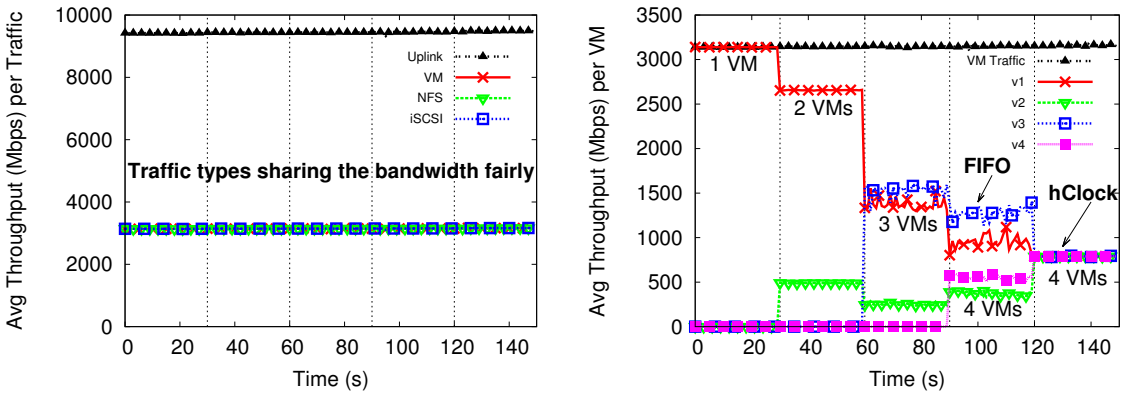

Figure 9. (a) Performance of NFS, iSCSI and VM traffic type (b) Performance of VM workloads within VM traffic class

four VMs, v_1, v_2, v_3 and v_4 for the VM traffic class. All VMs issued network traffic using different configurations in $netperf$ benchmark as described in Table 3 where v_{nfs} and v_{iscsi} VMs map to v_5 and v_6. The names v_{nfs} and v_{iscsi} are mainly used for exposition here and the results are expected to be similar if we use actual IO traffic. The scheduling hierarchy used initially is depicted in Figure 8(a).

At $T = 0s$, v_1, v_{nfs} and v_{iscsi} are the only active contenders on the uplink capacity. Figure 9(a) shows the overall allocation for each traffic class. Each class is able to get $\frac{1}{3}$ of the bandwidth capacity based on the equal ratio set earlier. This helps the network administrators meet their first isolation goal.

From $T = 30s$, the other VMs namely v_2, v_3 and v_4 are powered on one after another with a gap of $30s$. While the three bottom traffic classes are scheduled based on shares, due to the flat nature of SFQ the flows within each class are scheduled in a FIFO manner while sharing a single queue. Figure 9(b) clearly shows that every new VM adds significant variance in throughput stability within the VM traffic class. This phenomenon is due to the latency jitter incurred by competing flows on the same queue, which we also call as noisy neighborhood problem.

One possible solution for this problem is to create a class of service with shares for every single VM under the root node. However as explained in Section 2 such flat model is not convenient.

A better and more elegant solution to this problem is to use the aggregation property of a hierarchical scheduler. hClock effectively introduces a queue Q_i for each v_i as shown in Figure 8(b). While these queues could have any shares, reservation and limit settings, a default shares value applied uniformly across them is enough to address this noisy neighborhood problem.

At $T = 120s$ in Figure 9(b), we replace SFQ with hClock algorithm. The hierarchy given to hClock depicted by Figure 8(b) provides the NFS, iSCSI and VM traffic types with equal shares as well as equal shares for v_1, v_2, v_3 and v_4 within the VM traffic class. The allocation after $T = 120s$, demonstrates that hClock is able to preserve the previous setting among the traffic types, while allocating a fair share to all VMs: v_1, v_2, v_3 and v_4. Now an admin can add more VMs in to the system without worrying about the change in allocation to other traffic types. Hierarchical model can also be used in a multi-tenant environment by grouping the VMs from the same tenant together and allocating bandwidth to them in an aggregated manner.

6.3 Diverse Controls Settings for Dynamic Workloads

Next we arranged the six VMs in Table 3, in the hierarchy shown in Figure 10 and evaluated reservation, limit and shares controls at various levels in the hierarchy.

For each experiment, we present both the expected value and the observed value of bandwidth allocation. It is worth nothing that the expected bandwidth allocation is computed based on a line rate equal to $\simeq 9400$ Mbps consistent across the systems used for our experiment. In reality this line rate varies between 9400 Mbps and 9500 Mbps which explains some of the slight discrepancies between expected and observed bandwidth allocation presented below. Finally, for readability purposes we rounded up the entitlements computed below to the closest multiple of 5.

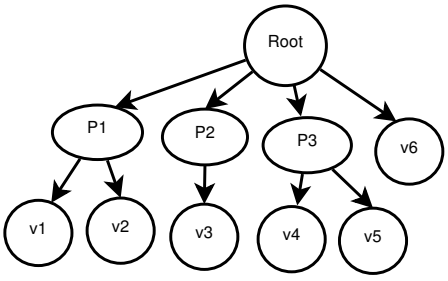

Figure 10. Scheduling hierarchy used for experiments.

6.3.1 Shares-Based Bandwidth Allocation

At $T = 0s$, we start with only the shares settings in the hierarchy shown in Table 4. Based on the share values, p_2

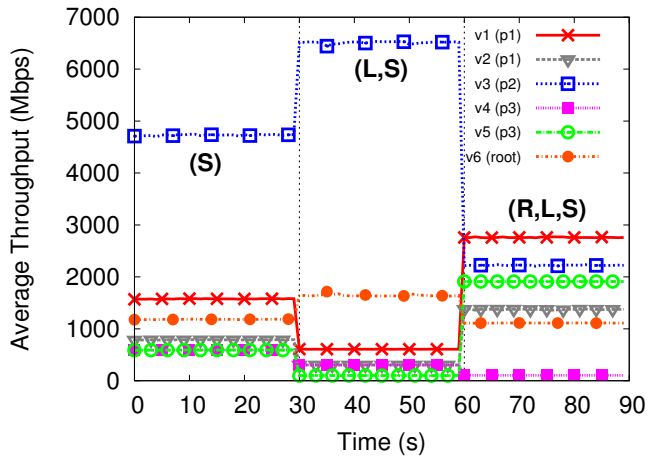

Figure 11. Observed bandwidth allocation (in Mbps) for VM workloads from Table 3. Every 30 seconds interval corresponds to Section 6.3.1, 6.3.2 and 6.3.3 respectively.

	R:L:S	Expected	Observed	Packets/s
p_1	0:∞:2	2350	2360	
v_1	0:∞:2	1565	1575	4490
v_2	0:∞:1	785	785	66300
p_2	0:∞:4	4700	4730	
v_3	0:∞:1	4700	4730	60240
p_3	0:∞:1	1175	1180	
v_4	0:∞:1	585	590	68960
v_5	0:∞:1	585	590	2270
v_6	0:∞:1	1175	1185	9020

Table 4. QoS settings along with expected and observed bandwidth (in Mbps). Packets/s shows the diversity in traffic (Section 6.3.1)

should get about $\frac{1}{2}$ of the available bandwidth and p_1 should get $\frac{1}{4}$. The remaining quarter is fairly distributed between p_3 and v_6. We chose to have v_6 as a direct child of root to show that the algorithm behaves properly regardless of the level at which the packets are actually queued up.

Table 4 shows that the observed bandwidth allocation for various nodes is within 1% of their expected entitlement. This happens at all levels in the tree. For instance, within p_1, the children v_1 and v_2 have share in ratio $2:1$ which is also respected. Figure 11 shows the individual allocations over time from $T = 0s$ to $T = 30s$.

6.3.2 Maximum Bandwidth Enforcement

Next we evaluated the limit enforcement of hClock . Table 5 shows the QoS settings applied to the scheduling hierarchy at $T = 30s$.

Hierarchical scheduling is very much about aggregation so we decided to first enforce a maximum constraint on p_1 while not capping any of its children. Previously, p_1 was entitled a bandwidth of 2350 Mbps and we limit it to 900 Mbps. Referring again to Table 5, we not only see the overall

	R:L:S	Expected	Observed	Packets/s
p_1	0:900:2	900	910	
v_1	0:∞:2	600	607	1740
v_2	0:∞:1	300	305	25710
p_2	0:∞:4	6480	6520	
v_3	0:∞:1	6480	6520	75740
p_3	0:400:1	400	405	
v_4	0:∞:1	300	305	35710
v_5	0:100:1	100	100	510
v_6	0:∞:1	1620	1650	12440

Table 5. QoS settings along with expected and observed bandwidth (in Mbps). Packets/s shows the diversity in traffic (Section 6.3.2)

	R:L:S	Expected	Observed	Packets/s
p_1	3000:∞:2	4100	4130	
v_1	0:∞:2	2735	2755	7870
v_2	0:∞:1	1365	1375	116290
p_2	0:∞:4	2200	2220	
v_3	0:∞:1	2200	2220	27700
p_3	2000:2000:1	2000	2014	
v_4	0:∞:1	100	104	12000
v_5	1800:∞:1	1900	1910	7340
v_6	0:∞:2	1100	1110	8370

Table 6. QoS settings along with expected and observed bandwidth (in Mbps). Packets/s shows the diversity in traffic (Section 6.3.3)

bandwidth allocation of p_1 capped at 900 Mbps but we also see this limited bandwidth divided between v_1 and v_2 in a $2:1$ ratio.

Then, we put a cap of 400 Mbps on p_3 as well as a cap of 100 Mbps on v_5. The expected bandwidth allocation for p_3 is naturally 400 Mbps since its shares based allocation provided much more than that. Finally, v_5 should only get 100 Mbps, providing the remaining 300 Mbps to its sibling. Figure 11 shows that within the time frame from $T = 30s$ to $T = 60s$ the observed bandwidth allocation is about 1% away from expectation. Note that the bandwidth taken away from p_1 and p_2 is not wasted and is instead allocated to the remaining contenders, namely v_3 and v_6, based on shares.

6.3.3 Minimum Bandwidth Guarantee

After demonstrating the efficacy of hClock at dealing with shares and limits on a hierarchy, we shift our focus to reservation. Table 6 shows the overall QoS settings applied to the hierarchy at $T = 60s$.

According to shares, p_1 and p_3 should only get around 2350 Mbps and 1175 Mbps respectively, as shown in Section 6.3.1. In this scenario, we decided to crank up their entitlement by configuring high reservations. First, we set p_1 to get a at least 3000 Mbps without setting any reservation for its children. Second, we would like p_3 to get at least 2000 Mbps but no more than that therefore we added a maximum

constraint equal to the reservation. Within p_3, we added a reservation of 1800 Mbps for v_5.

While the shares semantic being used does not matter when using a mix of shares and limit only since both would yield the same outcome, it becomes extremely relevant once reservations come into play. In this particular experiment, we decided to use the SUM(R,S) semantic. We will show in the next section that hClock can handle both semantics with a simple configuration change.

Starting with p_3, its expected bandwidth allocation is 2000 Mbps, since its minimum and maximum guarantee are equal. Going one level down, v_5 gets 1800 Mbps of reservation and the spare bandwidth of 200 Mbps is split equally based on shares and the SUM(R,S) semantics.

Moving to p_1, it should get at least 3000 Mbps and some more based on its share of the spare bandwidth. With p_1 and p_3 together reserving 5000 Mbps, the spare bandwidth adds up to 4400 Mbps. Therefore with p_3 not eligible for any additional bandwidth, the total number of shares comes to 8. Hence, p_1, p_2 and v_6 should get respectively $\frac{1}{4}$, $\frac{1}{2}$ and $\frac{1}{4}$ of the available spare bandwidth, which respectively turns out to be 1100 Mbps, 2200 Mbps and 1100 Mbps. While this is the final bandwidth allocation for p_2 and v_6, p_1 needs to add this to its already claimed reservation bringing up its entitlement to 4100 Mbps which in turn is divided between v_1 and v_2 in a $1:2$ ratio, based on their shares.

Table 6 summarizes the expected bandwidth allocation for each node in the hierarchy and shows, along with Figure 11, from $T = 60s$, that hClock is able to deliver the full set of QoS controls **(R,L,S)** in an hierarchical manner while dealing with very heterogeneous networking workloads.

6.3.4 Different Shares Semantics

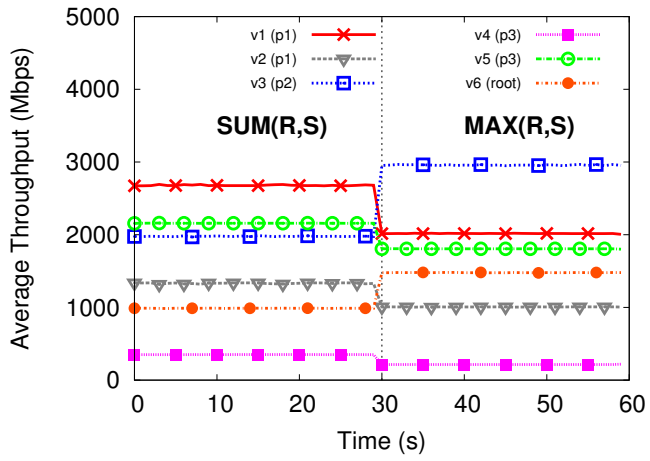

Figure 12. Observed allocation (Mbps) of VMs using **SUM(R,S)** from $T = 0$ to 30 sec and **MAX(R,S)** afterward.

As mentioned earlier, many network administrators prefer the SUM(R,S) semantics and find it easier to grasp. However some administrators prefer the MAX(R,S) shares

	$R{:}L{:}S$	SUM(R,S)	MAX(R,S)
p_1	3000:∞:2	3980 / 4005	3000 / 3024
v_1	0:∞:2	2655 / 2675	2000 / 2016
v_2	0:∞:1	1325 / 1330	1000 / 1008
p_2	0:∞:4	1955 / 1975	2935 / 2955
v_3	0:∞:1	1955 / 1975	2935 / 2955
p_3	2000:∞:1	2490 / 2510	2000 / 2020
v_4	0:∞:1	345 / 352	200 / 214
v_5	1800:∞:1	2145 / 2158	1800 / 1806
v_6	0:∞:2	980 / 988	1465 / 1478

Table 7. QoS settings along with Expected / Observed bandwidth (in Mbps) for different share semantics

semantic offered in VMware ESX Server hypervisor [27] for CPU and Memory scheduling today. In this experiment we show that hClock can easily switch from one semantic to another. Table 7 depicts the QoS settings applied to the scheduling hierarchy.

At $T = 0s$, the shares semantic SUM(R,S) is deployed. After removing the reserved bandwidth adding up to 5000 Mbps, the spare bandwidth of 4400 Mbps is divided based on shares. Practically, it means that p_1, p_2, p_3 and v_6 will share the spare bandwidth in ratio $2:4:1:2$, which in turn will be further divided among their children. Table 7, column SUM(R,S) and Figure 12 clearly show that the observed bandwidth during the time frame starting at $T = 0s$ to $T = 30s$ matches our computation.

At $T = 30s$, we changed the shares semantic to MAX(R,S). Looking at p_1, it has a minimum requirement of 3000 Mbps which is higher than its shares based allocation of $\frac{2}{9}$ of 9400 Mbps = 2090 Mbps. Similarly p_3 has 2000 Mbps of reservation but the allocation based on its shares only adds up to 1045 Mbps. Basically it means that p_1 and p_3 are already exceeding their entitlement based on shares and will not be eligible for any additional bandwidth. Therefore, p_2 and v_6 will split the remaining unclaimed 4400 Mbps in ratio $2:1$. The same logic applies for the children of p_3 where v_5 has a reservation of 1800 Mbps, which is much higher than its shares based allocation. So v_4 should get all of the remaining 200 Mbps. Table 7, column MAX(R,S) and Figure 12 show that the allocated bandwidth match very closely to the expected values after $T = 30s$.

6.4 hClock : CPU Overhead Comparison

In this section, we compare the CPU utilization of hClock with less sophisticated scheduling disciplines. We compared three different scheduling variants: FIFO, proportional sharing (emulated by using a flat hierarchy with shares control only, using hClock) and a two-level hierarchy configured with (R,L,S) controls at each node using hClock . The two-level hierarchy consisted of 10 internal nodes at the first level each with 10 children to accommodate a total of 100 VMs. The VMs produced a total steady rate of 60K packets/sec.

Figure 13 shows the normalized utilization of single CPU

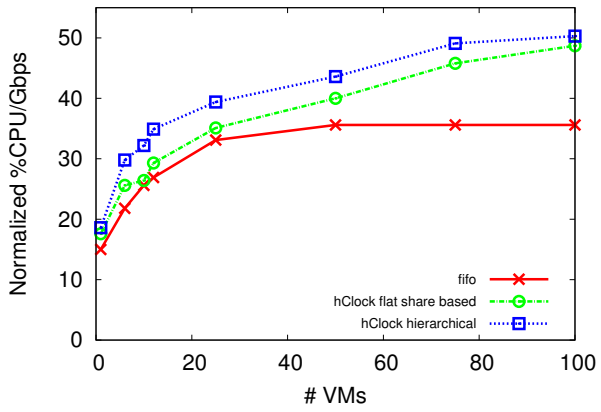

Figure 13. Normalized %CPU/Gbits for different number of VMs while using (a) FIFO, (a) hClock with no hierarchy, only shares and (c) hClock with a two-level hierarchy

per Gbps of transmit bandwidth as the number of VMs increases. The normalized CPU overhead between a flat hierarchy with shares and a two-level hierarchy using hClock seems to be within 5% while increasing up to 15% against the FIFO case. After further profiling, we discovered that the difference of 15% is not necessarily due to hClock complexity but instead is a side effect of the fairness provided by the algorithm.

In case of FIFO, packets from a given stream can easily be sent together in a burst. However, hClock does a good job of interleaving packets from the VMs to provide performance predictability and less jitter. This results in a higher number of TCP ACK packets from the receiver because the TCP delayed ACK timer timeouts more frequently under less bursty traffic. This higher number of TCP ACKs led to a higher interrupt rate. After profiling the CPU utilization of the ESX server and the guest VMs, we observed that the majority of the CPU overhead was due to the time spent in servicing those extra interrupts to the guest VMs.

6.5 Overcommitted Reservation

One of the final key question we need to handle is how much capacity to reserve and how to handle over-committed scenarios where the total reservation exceeds the total available capacity? This can happen irrespective of admission control due to the nature of variability in throughput. For instance, an upstream switch might issue some pause notifications while trying to address some severe contention or a faulty hardware might force the link speed to go down. More commonly, insufficient hardware capabilities might impact the ability to reach line rate. The PCI subsystem being used is typically one of the culprits.

In practice, we find that reserving no more than 75% of the total link capacity is a safer option. Beyond this utilization, the link performance might not be as determistic as expected due to the reasons mentioned earlier. However, we definitely need to handle the capacity fluctuations.

In overcommitted situations, hClock is designed so that reservations take over and the bandwidth allocation is done based on relative ratio of reservations. We forced such an environment by setting a high reservation value at each node in the hierarchy to get an overall 16 Gbps reservation on a 10 Gbps link and verified that the allocation falls back to the ratio of reservation values. For this reason we suggest that every flow should have a small reservation value to avoid starvation.

Interestingly, we noticed that in an overcommited environment, the reservation tag of VMs is always lagging behind the physical time due to the delay in meeting reservations. Normally this is not an issue as long as everybody stays active since the R_q will be served in the order that follows the reservation based ratio. An issue however appears when one of the VM becomes idle for some time. When it becomes active again, hClock will adjust the R_q of the VM to the current time T as discussed in Section 3. Since other VMs are running behind, the newly active VM will be denied service until the other active VM's R_q catches up to T.

This starvation issue can be addressed by noting that a VM becoming active with a lagging R_q should be eligible as soon as possible. Regardless whether or not the system is over-committed, being eligible right away means aligning R_q with the minimum between the current time T and the minimum reservation tag (R_{min}) in the system at synchronization point, using the following equation:

$$R_q \leftarrow \max\{R_q, \min\{T, R_{min}\}\} \tag{9}$$

Using the equation above, our experiments showed that hClock is able to handle the over-committed cases with dynamic workloads.

7. Conclusions and Future Work

In this paper, we studied the problem of providing richer QoS controls for network bandwidth allocation in a hypervisor to handle various use cases in a cloud environment. We propose a novel algorithm hClock that enforces hierarchical reservation, limit and proportional shares controls by using a set of real-time and virtual-time tags at each level. We also presented several optimizations to reduce CPU overhead, and increase parallel execution. Finally we showed that our hClock prototype implementation in VMware ESX server hypervisor is able to efficiently enforce these controls for a diverse set of workloads.

While hClock role is not to provide by itself an end-to-end QoS solution in a datacenter, we believe that it is a solid building block at the edge toward this goal. Investigating how hClock could fit into an end-to-end QoS solution is part of future work.

Acknowledgments

We would like to thank our shepherd Jinyang Li and other reviewers for very insightful comments and suggestions. That substantially improved the overall presentation and quality of the paper. We also are very grateful to Jin Heo and Amitabha Banerjee from VMware performance team, who worked tirelessly in evaluating various performance aspects of hClock and helped us improve the efficiency of our implementation.

References

[1] J. C. R. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *Proc. of INFOCOM '96*, pages 120–128, March 1996.

[2] J. C. R. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5): 675–689, 1997.

[3] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three CPU schedulers in Xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2), 2007.

[4] F. Chiussi and A. Francini. Minimum-delay self clocked fair queueing algorithm for packet-switched networks. In *INFOCOMM'98*, 1998.

[5] R. L. Cruz. Quality of service guarantees in virtual circuit switched networks. *IEEE Journal on Selected Areas in Communications*, 13(6):1048–1056, 1995.

[6] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. *Journal of Internetworking Research and Experience*, 1(1):3–26, September 1990.

[7] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *SOSP*, 1999.

[8] S. Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proc. of INFOCOM*, 1994.

[9] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: communication-aware CPU scheduling for consolidated xen-based hosting platforms. In *VEE*, 2007.

[10] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Trans. Netw.*, 5(5), 1997.

[11] A. Gulati, I. Ahmad, and C. Waldspurger. PARDA: Proportionate Allocation of Resources for Distributed Storage Access. In *Proc. Conference on File and Storage Technology (FAST '09)*, Feb. 2009.

[12] A. Gulati, A. Merchant, and P. J. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[13] A. Gulati, G. Shanmugnathan, X. Zhang, and P. Varman. Demand Based Hierarchical QoS Using Storage Resource Pools. In *Usenix Annual Technical Conference (ATC '12)*, June 2012.

[14] Hewlett Packard, Inc. HP LeftHand P4000 Storage. 2012. `http://www.hp.com/go/storage`.

[15] Nutanix, Inc. *The SAN free datacenter*. 2012. http://www.nutanix.com.

[16] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Trans. Netw.*, 1(3): 344–357, 1993.

[17] A. K. Parekh and R. G. Gallagher. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Trans. Netw.*, 2(2): 137–150, 1994.

[18] H. Sariowan, R. L. Cruz, and G. C. Polyzos. Scheduling for quality of service guarantees via service curves. In *Proceedings of the International Conference on Computer Communications and Networks*, pages 512–520, 1995.

[19] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *Proc. of SIGCOMM*, 1995.

[20] Simplivity, Inc. *The Simplivity Omnicube Global Federation*. 2012. http://www.simplivity.com.

[21] V. Soundararajan and J. M. Anderson. The Impact of Management Operations on the Virtualized Datacenter. In *ISCA*, 2010.

[22] D. Stiliadis and A. Varma. Efficient fair queueing algorithms for packet-switched networks. *IEEE/ACM Transactions on Networking*, 6(2):175–185, 1998.

[23] D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking*, 6(5):611–624, 1998.

[24] I. Stoica. Stateless Core: A Scalable Approach for Quality of Service in the Internet PhD. Dissertion, CMU-CS-00-176, 2000.

[25] I. Stoica, H. Zhang, and T. S. E. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services. *IEEE/ACM Trans. Netw.*, 8(2):185–199, 2000.

[26] S. Suri, G. Varghese, and G. Chandramenon. Leap forward virtual clock: A new fair queueing scheme with guaranteed delay and throughput fairness. In *INFOCOM'97*, April 1997.

[27] VMware, Inc. *Introduction to VMware Infrastructure*. 2007. http://www.vmware.com/support/pubs/.

[28] VMware vSphere 5. *Network I/O Control*. 2012. http://www.vmware.com/products/datacenter-virtualization/vsphere/network-io-control.html.

[29] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Dec. 2002.

[30] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu. vSlicer: latency-aware virtual machine scheduling via differentiated-frequency CPU slicing. In *HPDC*, 2012.